Shared Memory

OS Lecture 9

UdS/TUKL WS 2015

Review: Virtual Memory

How is *virtual memory* realized?

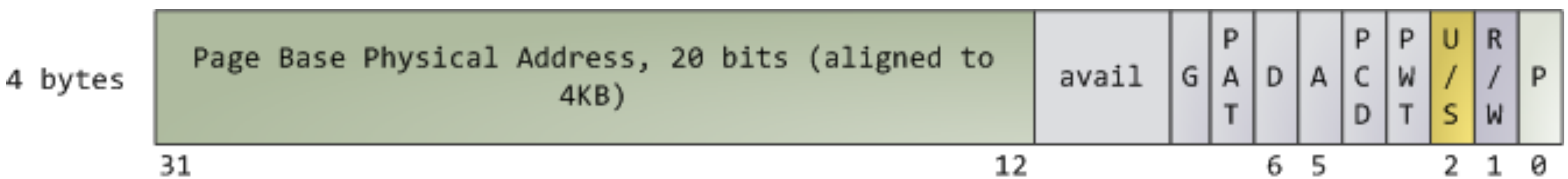
1. **Segmentation**: *linear* virtual ➞ physical address translation with *base* & *bounds* registers

2. **Paging**: *arbitrary* virtual ➞ physical address translation by lookup in page table

3. **Segmentation + paging**: first segmentation, then lookup in page table

   - virtual address ➞ *linear* address ➞ physical address
   - e.g., used in Intel x86 architecture (32 bits)

Example:[1] x86 Page Table Entry (PTE)



P: present **D: dirty** **A: accessed**

R/W: read or read+write

U/S: user or supervisor (kernel)

PCD: cache disabled PWD: cache write through

PAT: extension

---

[1] Figure from http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/ (A nice, easy-going tutorial; recommended further reading.)

Review: Sparse Address Spaces (1/2)

Why do we need explicit support for *sparsely populated* virtual address spaces? (= big "empty" gaps in virtual address space)

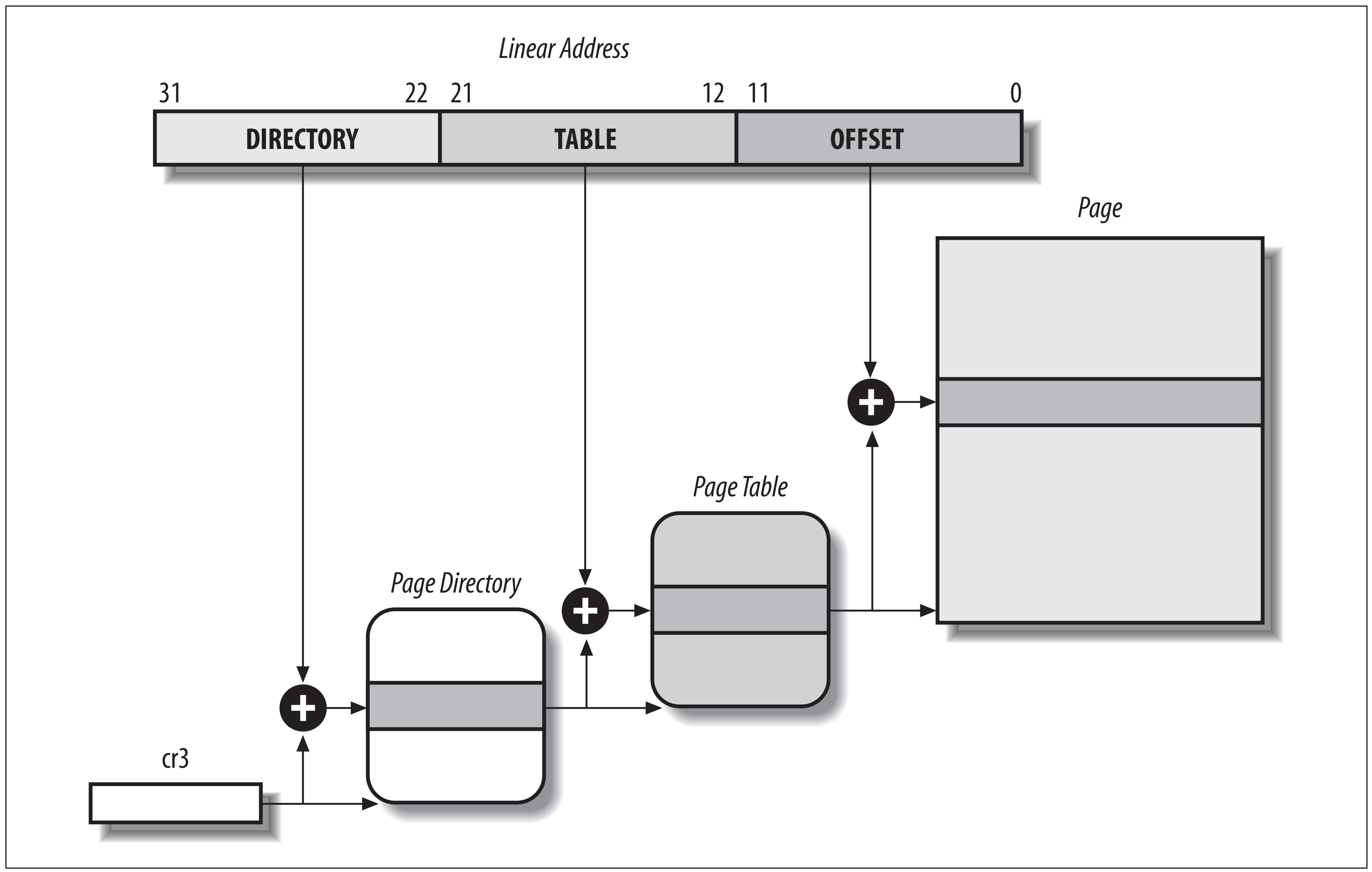
- Holes of unmapped addresses arise naturally due to shared libraries, kernel memory (at high memory), heap allocations, dynamic thread creation, etc.
- Problem: a *flat* page table can waste large amounts of memory
- Example: to represent $\mathbf{2^{32}}$ bytes = 4Gb of memory with 4Kb pages, we need $\mathbf{2^{32}/4096 = 1,048,576}$ PTEs
  - At 4 bytes (1 = word) per PTE,
    that's 1024 pages = 4Mb!

Review: Sparse Address Spaces (2/2)

How are *sparsely populated* virtual address spaces supported?

- Problem with flat page tables: most PTEs are marked invalid to represent “holes”
- Idea: represent “holes” *implicitly* by *absence* of PTEs, not *explicitly* with invalid PTEs
- Solution: *hierarchical* page tables: have **many shorter page tables**, use some bits of virtual address to look up which *page table* to use in a *page table directory*.

Example:[2] x86 Multi-level Page Table



[2] Figure from Bovet and Cesati, Understanding the Linux Kernel, O Reilly Media, 3rd edition, 2005.

Review: Missing Page Table Entries

What happens when a virtual address *cannot be resolved* by the MMU?

» “cannot be resolved” = either entry in page table directory is marked *invalid*, or PTE in page table is marked *invalid* (= not present)

» The result is a **page fault**: an exception is triggered and control is transferred to the OS-provided **page fault handler**.

» Page fault handler has access to all register contents, *faulting instruction*, and can implement *arbitrary policy*.

How do page faults differ from system calls?

And from other exceptions or interrupts?

- In large parts system calls, exceptions/traps, and interrupts are the same.
  - control flow diverted to OS-provided handler; processor switchees to kernel mode; register contents and *status code* provided
- Key difference: after *system call* or *interrupt*, resume execution at **next instruction**, but after *page fault*, **re-execute faulting instruction**

Exception during exception handling

What happens if a page fault (or any other exception/trap) is encountered while handling a page fault (or any other exception/trap)?

- On x86, a **double fault exception** (0x8) is generated, for which the OS must provide an exception handler.
- What happens if an exception is encountered while handling a *double* fault?
- On x86, a **triple fault exception** is generated, which *immediately resets* the system.

Shared Memory

What does it mean for a page to be "shared"?

- **Multiple processes** can *read from* and/or *write to* the same physical page.
- Historic platforms: all *physical* memory shared
  - any thread can read / write any memory location
- With segmentation / paging: no *virtual* memory shared at all: → all processes perfectly isolated
- But *selective* sharing is useful. How to re-enable it?

How to give access to a page of memory?

What does the OS have to do to share a page *P* of memory?

- Simply **insert a page table entry** (PTE) for the shared physical page in the page table of each process that shares *P*
- Any number of PTEs in any number of page tables can refer to the same physical page
- ***Same*** *physical page* can be mapped by different processes *at* ***different*** *virtual addresses*
  - beware of pointers in shared memory segments!

How to take away access to a page of memory?

What does the OS have to do to “un-share” a page of memory?

» **Remove PTE** (= mark as *non-present*) in the page table of process that loses access rights.

» Is this enough?

» No! Stale mapping could still exist in *translation look-aside buffers* (TLBs) of one or more cores

Review: When to flush the TLB?

- When introducing a **new mapping** — adding a PTE to the page table at a previously invalid virtual address — **no TLB flush** is required.
- When changing an **existing mapping** — overwriting a valid PTE — a **TLB flush** is required: a *stale TLB* entry may exist.
- When **removing a mapping** — zeroing a valid PTE — a **TLB flush** is required.
- What happens on multiprocessors?

When and why does the OS share memory?

1. **Explicitly**, when requested by applications
   - To enable efficient *communication*
2. **Implicitly**, to optimize resource usage
   - Memory is scarce and valuable,
     must be used efficiently
   - This happens *transparently* to applications

Explicitly Shared Memory (1/2)

Example: In POSIX, user process can request a *shared memory segment* with `mmap()`.

```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- `addr` — where to map the memory in **virtual** address space
- `length` — how much to map (multiple of page size)
- `prot` — combination of `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, or `PROT_NONE`
- `flags` — `MAP_SHARED` and many special cases…
- `fd` — file to map
- `offset` — offset within file where the mapping starts

Explicitly Shared Memory (2/2)

- **Access control**: two processes may (explicitly) share memory if and only if they can map the same file (*file system permissions* apply)
  - can create temporary files as needed
- **Backing pages**: file is represented in memory by (physical) pages anyway.
- **Application-controlled**: OS just installs / removes PTEs corresponding to requested operations.

Implicitly Shared Memory

Basic idea: store *redundant* information only once
Examples:

- **Multiple instances of the same program**, but only one *read-only* copy of **text segment** (code)
- …only one *read-only* copy of **constant data**
- **Shared library** used by many processes, but only one *read-only* copy of **library code and constants**

Can we share even more?

Memory is scarce and copying is expensive.
Can we *share additional memory*? Heap memory? Stack memory? Global variables?

- Problem: Heap, stack, globals are **writable** pages.
  - Naïve sharing of *writable* memory
    → processes overwrite each other's updates!
- But: *most* writable memory is *never* written to in a typical process.
  - Which memory is written to depends on input.

Copy-on-Write (CoW)

Idea: Need a new copy of a *writeable* page only each time it is actually written to.

- We can allocate such copies *lazily* on demand.
- When a write occurs, *transparently* make a copy of the *shared* page and give the new copy to the writing process, making it *private*.
- To do so, we must *trap* (= detect) a write attempt.
  - This can be accomplished with PTE protection bits…
- **Tradeoff**: Nothing gained if *all* pages are written to, but most programs modify only *some* of their memory.

How does CoW work?

1. Shared page is marked as *read-only* in page tables of all processes that share it.
   » OS must keep track of in which address spaces a physical page is mapped
2. As a result, any write attempt leads to a page fault.
3. When a process traps into the kernel due to a write attempt, a new physical page is allocated and a copy of the shared page is made.
4. The page table of the process that trapped is updated to point to the newly allocated page, which is mapped with *read-write* permissions.
5. The process that trapped is resumed by re-executing the faulting instruction.

Applications of CoW

Some examples where CoW can have great effect:

» In UNIX and UNIX-like systems, new processes are created with `fork()` by *duplicating* the calling process. The semantics of `fork()` require the entire address space to be “copied” — this is much faster with CoW.

» Shared libraries with rarely-changed defaults

» *Privately* mapped files: where changes by one process should not be seen by other processes.

Where does paged memory come from?

With *CoW* and *demand paging*, the page fault handler *lazily* sets up the page based on an *authoritative reference* page (e.g., file contents).

Generalizing this notion, does the authoritative source always have to be local?

» No! The page fault handler can determine contents of page *arbitrarily*. E.g., via a network.

Distributed Shared Memory

What if we want to write **multithreaded program** that takes advantage of **all cores** in a **cluster** connected by a *fast* network?

- We can create a *single* virtual address space that spans *separate* physical memories.
- The same virtual address space is used by threads on all hosts.
- Basic idea: map pages as always, but the underlying physical **reference page may reside on a remote host**.
- To make this work, transfer page contents as needed: **from** remote host's memory when *paging in*; **to** remote host's memory when *paging out* (or when *flushing* dirty pages).

Coordinating Writes in a DSM

How can a node safely write to a page that may be mapped on remote hosts?

» We can keep a **cache** of **read-only copies** of a page on *multiple* hosts simultaneously.

» When a **write traps** on any host, **invalidate** the page on all other hosts by **(a) evicting** it from their caches and **(b)** unmapping it from the virtual address spaces of their local processes.

» Once a host is the **exclusive owner** of page, it can allow the local process to write to it.

DSM relies on read-only and read-mostly pages

Remember: many pages are never, or only rarely, written to. Further, threads rarely access all pages. This is essential to making DSM work. Why?

» If all shared pages are written to, the speed of the computation will be limited by the network speed (to propagate updates).

» If each thread accesses most pages, these will not all fit into its local memory; the speed of the computation will be limited by the network speed as it pages in data from remote hosts.

DSM Principles in Modern Systems

DSM at the process level has fallen out of favor in current system design. But the basic principles are widely used nonetheless. Where and why?

» A modern multicore processor resembles a “distributed systems on a chip”.

» Multiple sockets, with multiple cores and shared caches each.

» Very fast local caches, much slower access to remote caches or global memory.

» **Cache consistency protocols**, operating at the level of cache lines, are conceptually very similar to page-based DSMs.

Common VM “Trick”: Guard Pages

How to catch *heap* or *global* buffer over- and underflows?

- Programs tend to write beyond allocated buffers…
  - e.g., off-by-one errors, wrongly computed bounds
- These can be **difficult to catch during testing**, but can have disastrous consequences (security vulnerabilities).
- **Guard pages**: between any two heap allocations, keep some unmapped virtual addresses
  - access past buffer = page fault = obviously an error

What to do with segments?

If we have both *paging* and *segmentation*, are segments still useful?

- Depends. Segments are not necessary to provide virtual memory.
  - E.g., Linux does not use segments to realize address spaces even if the hardware supports segmentation.
- However, there are other techniques for which segments can be useful.
  - Two examples here…

Another Defensive Technique: Stack Canaries

How to catch buffer overflows on the *stack*?

- **Security vulnerability**: buffer overflows on the stack can overwrite return address, hijack control flow.
- **Defense**: place *canary value* on stack before return address — before returning from function, check for overwritten canary value.
- But where to store reference value such that attacker can't get to it?
- One solution: in a *separate segment*, the position of which is randomized when the program is loaded.

How to Implement Thread-local Variables?

Thread-local variable:[TLS] *each thread* has a **private copy** of a variable, can be accessed without locks.

Per-CPU variable: in kernel, *each core* has a private copy of a variable. (Ex: scheduler queues)

» Same code for all cores / threads, but must reference different physical addresses. How?

» One approach: use *segment* for thread-local variables.

» Each core / thread uses different *base & bounds registers*

[TLS] Further reading: http://www.akkadia.org/drepper/tls.pdf